Deep Reinforcement Learning in Large Discrete Action Spaces

Maanik Arora (20161192)
Shashank Srikanth (20161103)
Nikhil Bansal (20161065)

IIIT Hyderabad

November 28, 2019

Overview

Problem Statement

- Reasoning in an environment with a large number of discrete actions (million) is important.
- Need to generalize over the set of actions in sub-linear complexity relative to $|\mathcal{A}|$
- Applications in Recommender Systems, Industrial Plants and Language Models.
- Previous algorithms (Value Func. Approximation, Policy Gradient) fail in large discrete action space environments.

Definitions and Notations

- $\mathcal{A}$: Set of discrete actions.
- $\mathcal{S}$: Set of discrete spaces
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is transitional probability.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$.

- Each action $\mathbf{a} \in \mathcal{A}$ is a $n-$dimensional vector.
- Each state $\mathbf{s} \in \mathcal{S}$ is a $m-$dimensional vector.
- $R_t = \sum_{i=1}^{T} \gamma^{i-t} r(s_i, a_i)$. is the discounted sum of rewards.

- The state-action value function $Q_\pi(s, a) = \mathbb{E}\left[R_1 | s_1 = s, a_1 = a, \pi\right]$ is the expected return starting from a given state s and taking an action a.
- In this paper, both Q and π are approximated by parametrized functions.
- $\pi_Q = \arg\max_{a \in \mathcal{A}} Q(s, a)$.

Problems with the current existing approaches

- Two main types of RL approaches:
  - Value Function Approximation
  - Policy Gradient algorithms

- One example of policy greedy relative to value function:

$$\pi_Q(\mathbf{s}) = \arg\max_{a \in \mathcal{A}} Q(\mathbf{s}, \mathbf{a})$$

Value function is often parametrized using S, A & $|\mathcal{A}|$ evaluations are required. Execution complexity grows linearly with $|\mathcal{A}|$ => Task becomes intractable

- In a standard actor-critic approach => policy is explicitly defined by a parameterized actor function:$\pi_\theta : \mathcal{S} \rightarrow \mathcal{A}$. π_θ is often a classifier-like function approximator.
- It scales linearly in relation to the number of actions and does not generalize well to unseen actions.

Proposed Idea

- The main idea behind is to leverage prior information about the actions to embed them in a continuous space upon which the actor can generalize.
- The policy produces a continuous action within this space, and then uses an approximate nearest-neighbor search to find the set of closest discrete actions in logarithmic time.

Wolpertinger Architecture

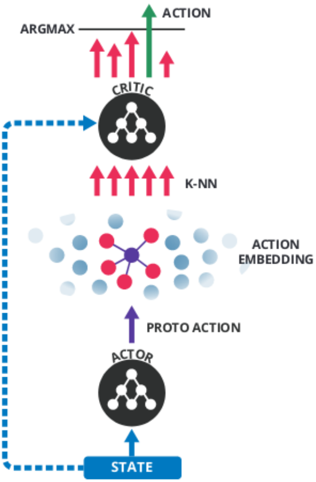

Figure: Wolpertinger Architecture

Proposed Approach

- New policy architecture called **Wolpertinger** architecture.
  - Avoids the heavy cost of evaluating all actions
  - Actor generates efficient action and Critic refines our actor's action choices
  - Retains generalization over actions and builds upon Actor-Critic framework
  - The policy is trained using Deep Deterministic Policy Gradient (DDPG).

Algorithm 1 Wolpertinger Policy

State $\mathbf{s}$ previously received from environment.
$\hat{\mathbf{a}} = f_{\theta^\pi}(\mathbf{s})$ {Receive proto-action from actor.}
$\mathcal{A}_k = g_k(\hat{\mathbf{a}})$ {Retrieve k approximately closest actions.}
$\mathbf{a} = \arg\max_{\mathbf{a}_j \in \mathcal{A}_k} Q_{\theta^Q}(\mathbf{s}, \mathbf{a}_j)$
Apply $\mathbf{a}$ to environment; receive $r, \mathbf{s}'$.

Figure: Wolpertinger Algorithm in brief

Action Generation

- The architecture reasons over actions within a continuous space R^n, and then maps this output (proto-action) to the discrete action space.

$$f_{\theta^\pi} : \mathcal{S} \to \mathbb{R}^n$$
$$f_{\theta^\pi}(\mathbf{s}) = \hat{\mathbf{a}}$$

- f_{θ^π} is a mapping from the state representation space $\mathbb{R}^m$ to the action representation space $\mathbb{R}^n$.
- This function (actor network) outputs a proto-action ($\hat{a}$) in $\mathbb{R}^n$ for a given state, which will likely not be a valid action.
- We map from $\hat{\mathbf{a}}$ to an element in $\mathcal{A}$ using K-Nearest neighbour algorithm. We can do this with:

$$g : \mathbb{R}^n \to \mathcal{A}$$
$$g_k(\hat{\mathbf{a}}) = \arg\min_{\mathbf{a}\in\mathcal{A}} \|\mathbf{a} - \hat{\mathbf{a}}\|_2$$

Action Refinement

- Actions with a low Q-value with respect to neighbouring actions might be the nearest neighbour of $\hat{a}$.
- Simply selecting the closest element to $\hat{\mathbf{a}}$ from the set of actions generated previously is not ideal.

The algorithm refines the choice of action by selecting the highest-scoring action according to Q_{θ^Q}:

$$\pi_\theta(\mathbf{s}) = \arg \max_{a \in g_k \circ f_{\theta^\pi}(s)} Q_{\theta^Q}(\mathbf{s}, \mathbf{a})$$

- Action refinement makes the algorithm significantly more robust to imperfections in the choice of action representation, and is essential in making the system learn in certain domains.

Complete Wolpertinger Algorithm

Algorithm 2 Wolpertinger Training with DDPG

```
Randomly initialize critic network Q_{θ^Q} and actor f_{θ^π} with weights θ^Q and θ^π.
Initialize target network Q_{θ^Q} and f_{θ^π} with weights θ^{Q'} ← θ^Q, θ^{π'} ← θ^π
Initialize g's dictionary of actions with elements of 𝒜
Initialize replay buffer B
for episode = 1, M do
    Initialize a random process 𝒩 for action exploration
    Receive initial observation state s_1
    for t = 1, T do
        Select action a_t = π_θ(s_t) according to the current policy and exploration method
        Execute action a_t and observe reward r_t and new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in B
        Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from B
        Set y_i = r_i + γ · Q_{θ^{Q'}}(s_{i+1}, π_{θ'}(s_{i+1}))
        Update the critic by minimizing the loss:
            L(θ^Q) = (1/N) Σ_i [y_i − Q_{θ^Q}(s_i, a_i)]^2
        Update the actor using the sampled gradient:
            ∇_{θ^π} f_{θ^π}|_{s_i} ≈
            (1/N) Σ_i ∇_a Q_{θ^Q}(s, â)|_{â=f_{θ^π}(s_i)} · ∇_{θ^π} f_{θ^π}(s)|_{s_i}
        Update the target networks:
            θ^{Q'} ← τθ^Q + (1 − τ)θ^{Q'}
            θ^{π'} ← τθ^π + (1 − τ)θ^{π'}
    end for
end for
```

Experiments

- Wolpertinger agent is evaluated on three environment classes:
  - Discretized Continuous Control
  - Multi-Step Planning
  - Recommender Systems

Discretized Continuous Control

- Each dimension d in the original continuous control action space is discretized into i equally spaced values, yielding a discrete action space with $|A| = i^d$ actions.
- Continuous control tasks like *Cartpole-v0* is used.
- Wolpertinger agent is able to reason with both small and large number of actions.

Multi-Step Plan Environment

- Environment has i actions available at each time step and planning n steps into the future means i^n possible actions
- Example: Puddle world environment - a grid world with four cell types: empty, puddle, start or goal.
- The goal of the agent is to find the shortest path to the goal that trades off the cost of puddles with distance travelled.
- The action set is the set of all possible n-length action sequence
- There are 2 base actions:$\{$down, right$\}$. This means that environments with a plan of length n have 2^n actions in total, for $n = 20$ we have $2^{20} \approx 1e6$ actions.

Recommender Environment

- A simulated recommendation system utilizing data from a large-scale recommendation engine was constructed.
- The environment is characterized by action set $\mathcal{A}$ and a transition probability matrix W
- $W_{i,j}$ defines the probability that a user will accept recommendation j given that the last item they accepted was item i.
- Each item has a reward r associated with it (if accepted by the user). The current state is the item the user is currently consuming, and the previously recommended items do not affect the current transition.

Evaluation

Evaluation of Wolpertinger algorithm is based on:

- Number of nearest neighbours (k)
  - k $= 1$ i.e immediate neighbour
  - k $= 0.5$ percent
  - k $= |\mathcal{A}|$
- Training time and average reward for full nearest-neighbor search, and three approximate nearest neighbor configurations. FLANN is used with three settings we refer to as ‘Slow’, ‘Medium’ and ‘Fast’.
  - **Slow:** Hierarchical k-means tree - 99% retrieval accuracy on the recommender task.
  - **Medium:** Randomized K-d tree 90% retrieval accuracy in the recommender task.
  - **Fast:** Randomized K-d tree - 70% retrieval accuracy in the recommender task.

Cartpole I

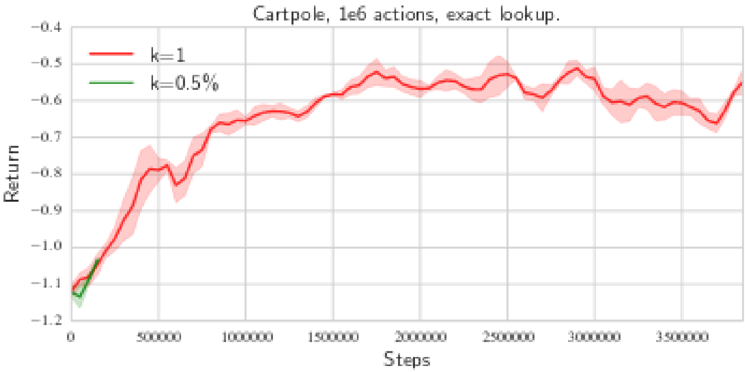

Figure: Agent performance for various settings of k with exact lookup as a function of steps

Cartpole II

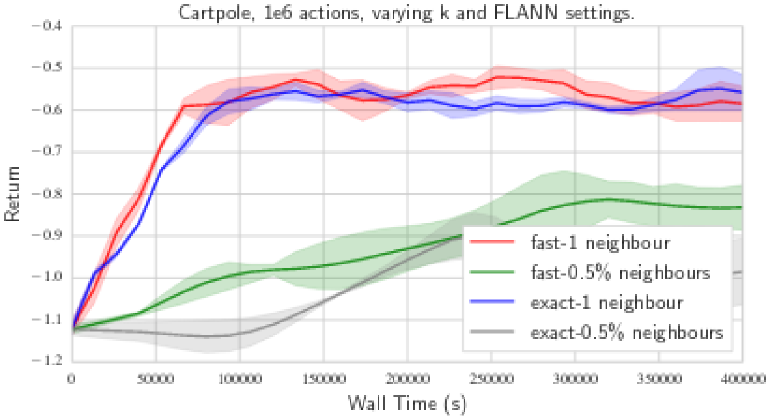

Figure: Agent performance for various settings of k and FLANN as a function of wall-time on one million action cartpole.

Cartpole III

# Neighbors	Exact	Slow	Medium	Fast
1	18	2.4	8.5	23
$0.5\% - 5,000$	0.6	0.6	0.7	0.7

Figure: Median steps/second as a function of k and FLANN settings on cart-pole.

Puddle World I

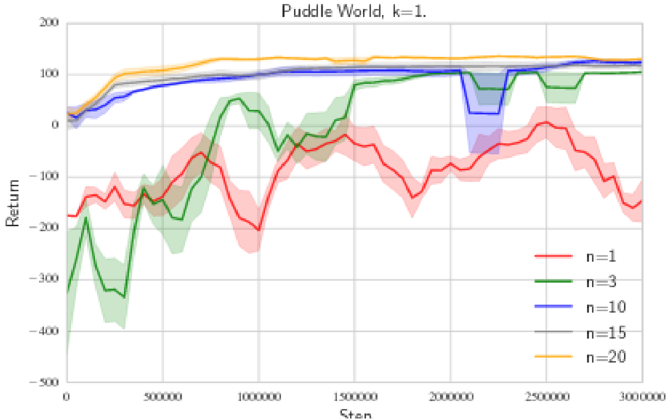

Figure: Agent performance for various lengths of plan, a plan of n= 20 corresponds to $2^{20} = 1,048,576$ actions

Puddle World II

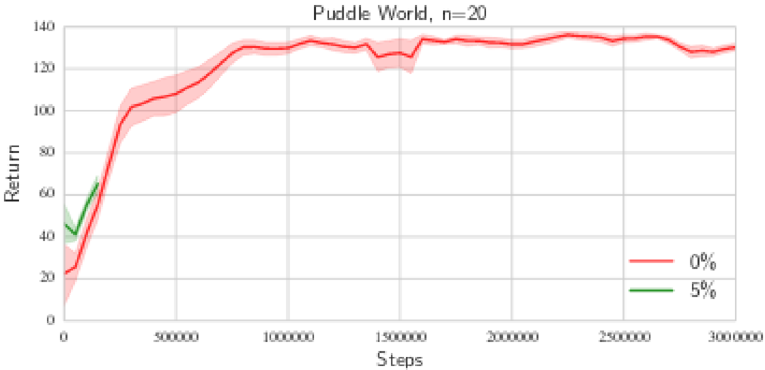

Figure: Agent performance for various percentages of k in a 20-step plan task in Puddle World with FLANN settings on 'slow'.

Puddle World III

# Neighbors	Exact	Medium	Fast
1	4.8	119	125
0.5% – 5,242	0.2	0.2	0.2
100% – $1e6$	0.1	0.1	0.1

Figure: Median steps/second as a function of k and FLANN settings.

Recommender Task I

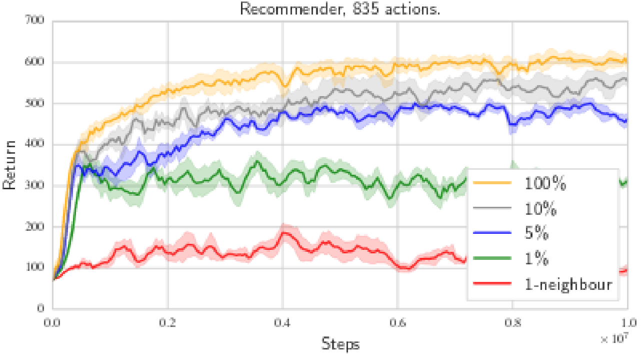

Figure: Performance on the 835-element recommender task for varying values of k, with exact nearest-neighbor lookup

Recommender Task II

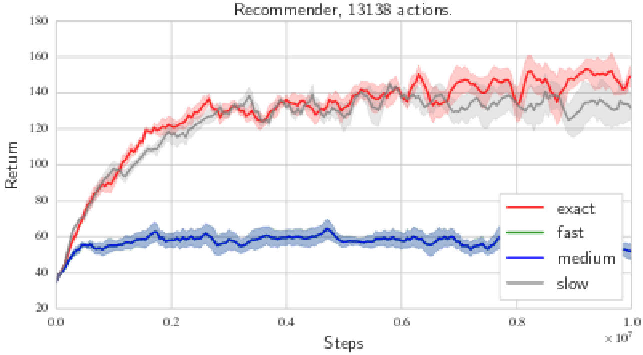

Figure: Agent performance for various numbers of nearest neighbors on 13k recommender task. Training with $k = 1$ failed to learn.

Recommender Task III

# Neighbors	Exact	Slow	Medium	Fast
1	31	50	69	68
1% – 131	23	37	37	37
5% – 656	10	13	12	14
10% – 1,313	7	7.5	7.5	7
100% – 13,138	1.5	1.6	1.5	1.4

Figure: Median steps/second as a function of k and FLANN set-tings on the 13k recommender task

Analysis I

- For a random proto-action $\hat{a}$, each nearby action has a probability p of being a bad or broken action with a low value of $Q(s, \hat{a}) - c$.
- The values of the remaining actions are uniformly drawn from the interval$[Q(s, \hat{a}) - b, Q(s, \hat{a}) + b]$, where $b \leq c$.
- The probability distribution for the value of a chosen action is therefore the mixture of these two distributions.

Lemma

Denote the closest k actions as integers $\{1, \ldots, k\}$. Then in the scenario as described above, the expected value of the maximum of the k closest actions is

$$\mathbb{E}\left[\max_{i \in \{1,\ldots,k\}} Q(s, i) | s, \hat{a})\right] = Q(s, a) + b - p^k(c - b) - \frac{2b}{k+1}\frac{1 - p^{k+1}}{1 - p}$$

Analysis II

- The highest value an action can have is $Q(s, \hat{a}) + b$. The best action within the k-sized set is thus, in expectation,$p^k(c-b) + \frac{2b}{k+1}\frac{1-p^{k+1}}{1-p}$ smaller than this value.
- The first term is in $O(p^k)$ and decreases exponentially with k.
- The second term is in $O(\frac{1}{k+1})$. Both terms decrease a relatively large amount for each additional action while k is small, but the marginal returns quickly diminish as k grows larger

Our Implementation

- As the given algorithm is similar to DDPG, we implement DDPG algorithm as well as WOLPERTINGER algorithm.
- To test our approach, we experiment on the Continuous cartpole environment provided by OpenAI Gym (Custom environment).
- We converted this continuous action environment into a discrete environment by dividing the continuous action space (-1, 1) into a million discrete actions in (0, 1).
- We use PyTorch for implementation and K-Nearest Neighbour algorithm is done PyFLANN algorithm

Resuts Obtained I

- Similar to the experiment methodology in the paper, we try our approach with multiple values of k.
- We specifically try k = 1, 10, 1000 and 100000.

Resuts Obtained II

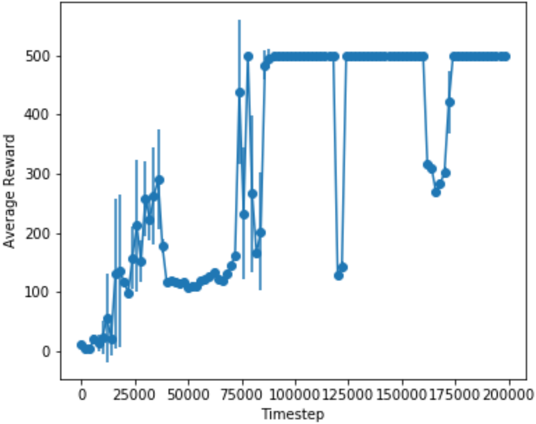

Figure: K = 1

Resuts Obtained III

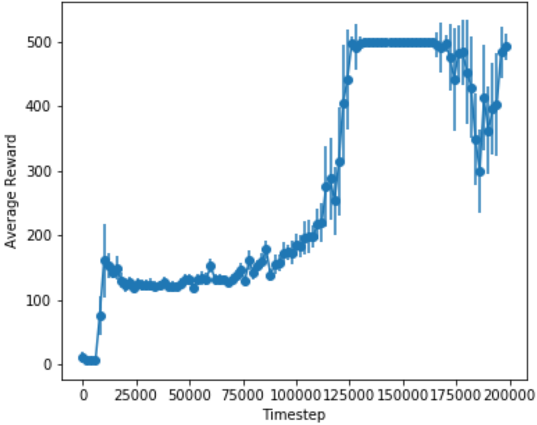

Figure: K = 10

Resuts Obtained IV

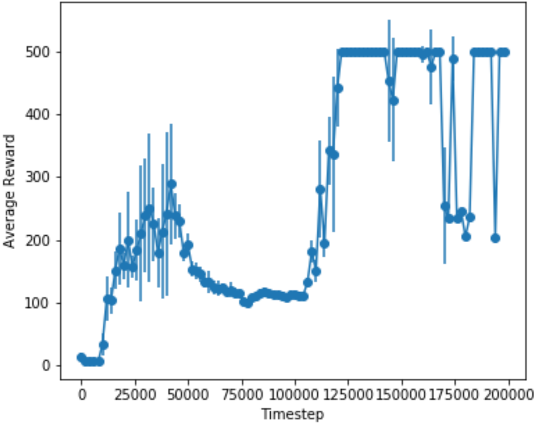

Figure: K = 1000

Conclusions I

- Our agent is able to learn the dynamics of the environment very well with mean reward of 500 after 150,000 steps, it consistently has a mean reward of 500 - maximum possible reward.
- Training is fastest in the case of $k = 1$ with high mean rewards, suggesting that for this task, it is the most suitable option of all the three.
- For $k = 100000$, the agent did not train at all as the computation time was too high, suggesting the efficacy of our approach.

Conclusions II

Number of neighbours (K)	Time Taken (minutes)
1	4.37
10	4.45
1000	6.24
100000	> 5 hours

Table 1: Time taken for 20,000 steps

- We also evaluated the various values of k for computational time by running the training loop for a maximum of 20,000 steps each.
- From the table, we can see that $K = 1$ is just marginally faster than than $K = 10$ but significantly faster than $K = 1000$.


ANY QUESTIONS???
meme-arsenal.ru